CT3536

GAMES PROGRAMMING

Andreas Ó hAoḋa
University of Galway
2023-11-03

Contents

1 Introduction

1.1 Lecturer Contact Information

- Dr. Sam Redfern (sam.redfern@nuigalway.ie)
- Discord server: https://discord.gg/nqD5JN95WT

1.2 Assessment

There will be 1-3 person projects which will begin from approximately Week 5, which will be worth 30% of the course marks. 10% of the overall mark will be from the six graded lab assignments, and the remaining 60% will be from the final exam.

The game project will have a demo in the final week, and will be submitted as a document containing images, discussions, code, etc. Marks are awarded for:

- Overall complexity.
- Code architecture & neatness.
- Game design/elegance & UX.
- Discussion.
- Graphics (if self-created).
- Audio (if self-control).
- Group size will also be taken into account.

You should start to consider your ideas by Week 5. Discuss the idea with Dr. Redfern & the lab tutors as they can advise on scope & difficulty. Simple 3D games are no harder to make than 2D games. "Snake" games are not allowd!'

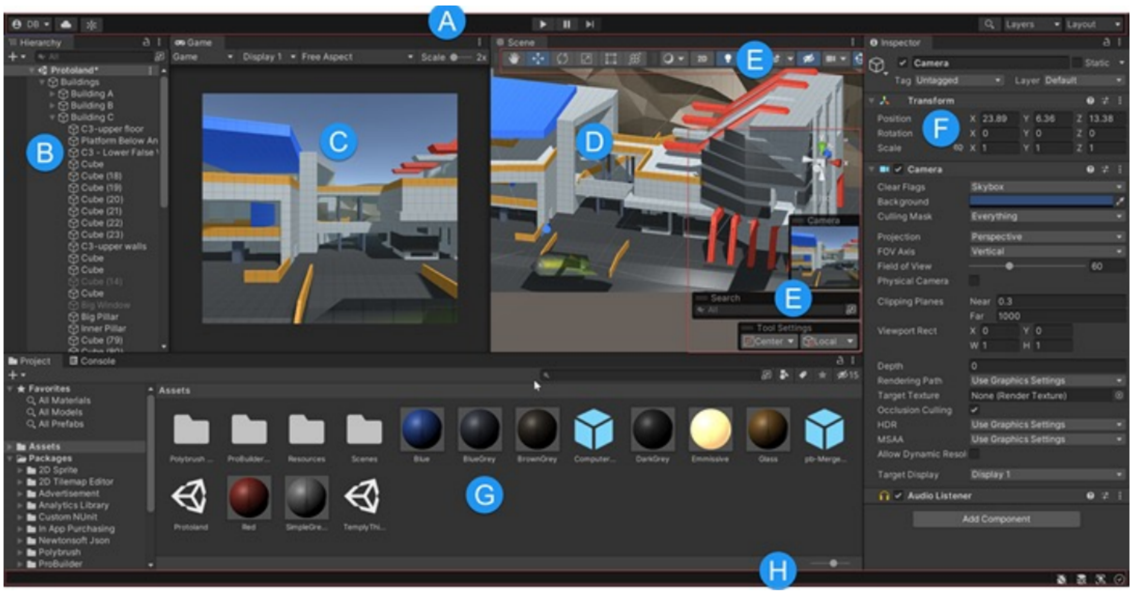
1.3 Game Engines

Game Engines provide a powerful set of integrated sub-systems geared towards making games (and other high-performance *realtime* media) including:

- Graphics Rendering (3D, 2D, terrain).
- Physics.
- Networking.
- Special Effects.
- 3D Audio.
- User Input.

1.4 Unity3D Game Engine

The Unity3D game engine is a closed-source games engine that is well designed & elegant to use "2$^{\text{nd}}$ generation" game engine. It has excellent GUI/HUD editing & animation system, and is powerful & very popular. It has a hugely successful Asset Store. Its core language is C#, which is very similar to Java. Unity3D deploys to iOS, Android, Web (HTML5/WebGL), Windows, Mac OSX, Linux, Switch, & more.



A. Toolbar	B Hierarchy Winow	C Game View
D Scene View	E Overlays	F Inspector Window
G Project Window	H Status Bar	

Figure 1: The Unity IDE

The **Project Window** displays your library of assets that are available to use in your project. When you import assets into your project, they appear here.

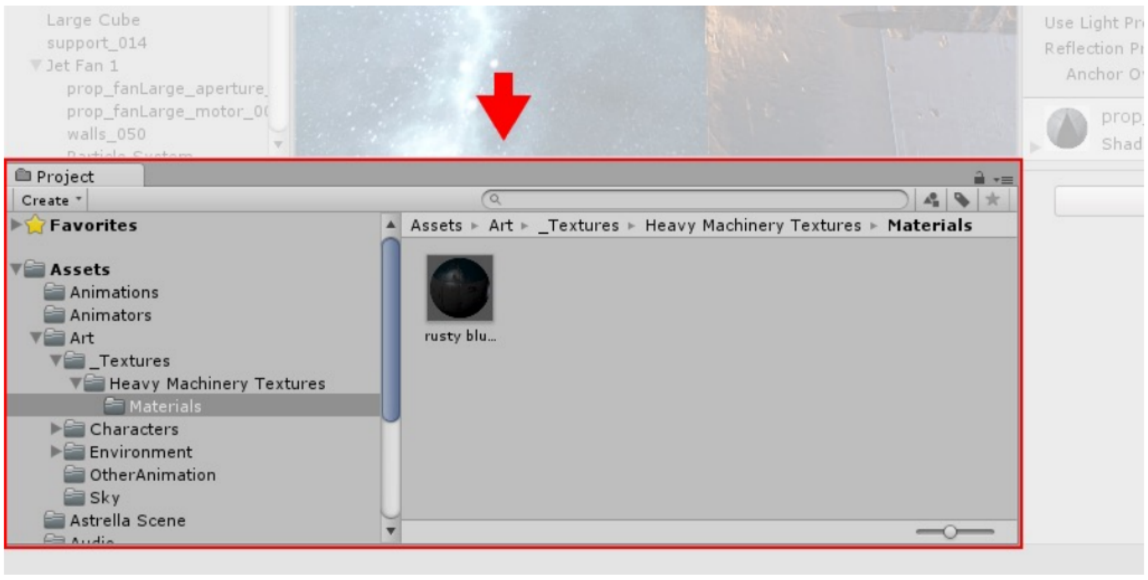


Figure 2: The Project Window (Assets)

The **Scene View** allows you to visually navigate & edit your scene without running your game. The scene view can be a 3D or 2D perspective.

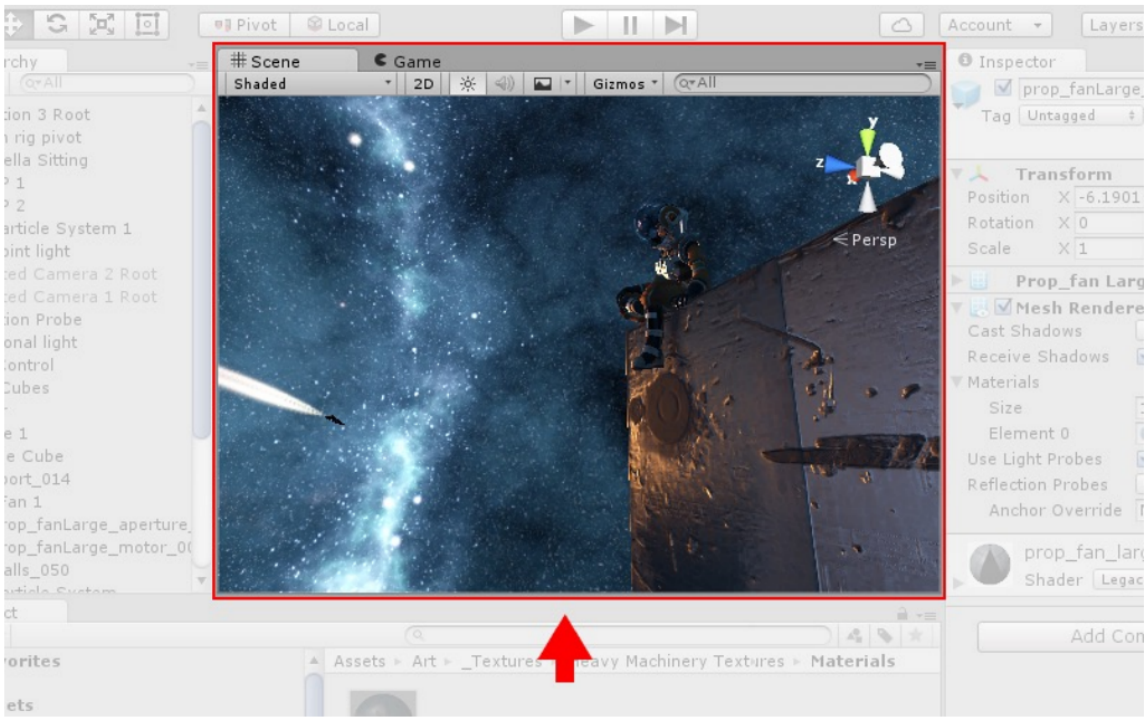


Figure 3: The Scene View

The **Hierarchy Window** is a hierarchical nested text representation of every "**game object**" in the scene.

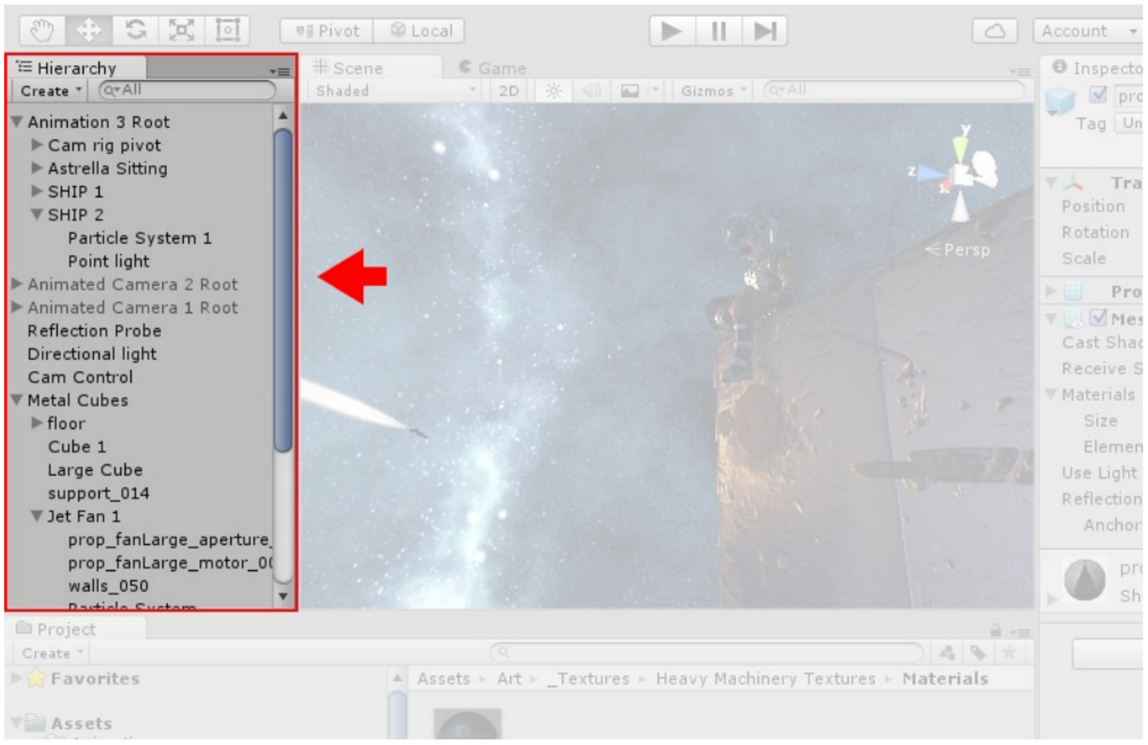
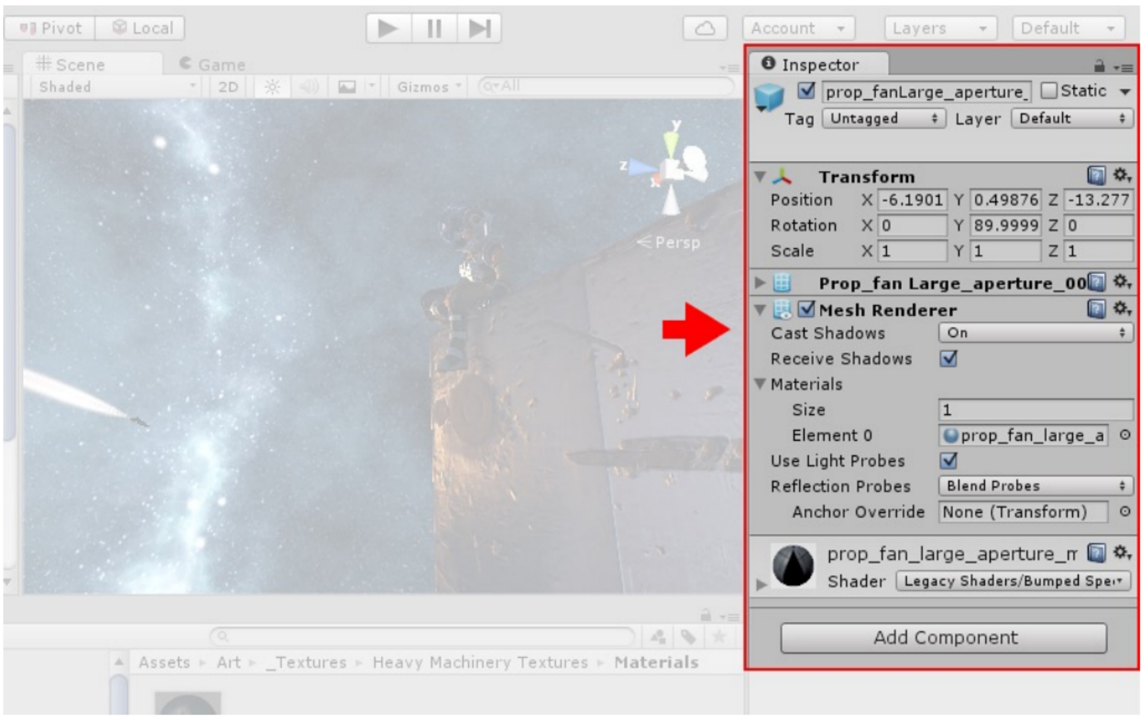


Figure 4: The Hierarchy (Scene Graph) Window

The **Inspector Window** allows you to view & edit all the properties of the currently selected game object.


Pivot
Local
Account
Layers
Default
Scene
Game
Shaded
2D
Gizmos
Persp
Assets ▸ Art ▸ _Textures ▸ Heavy Machinery Textures ▸ Materials

Inspector
prop_fanLarge_aperture_ Static
Tag Untagged Layer Default

Transform
Position X -6.1901 Y 0.49876 Z -13.277
Rotation X 0 Y 89.9999 Z 0
Scale X 1 Y 1 Z 1

Prop_fan Large_aperture_00
Mesh Renderer
Cast Shadows On
Receive Shadows ✓
Materials
Size 1
Element 0 prop_fan_large_a
Use Light Probes ✓
Reflection Probes Blend Probes
Anchor Override None (Transform)

prop_fan_large_aperture_m
Shader Legacy Shaders/Bumped Spe

Add Component

Figure 5: The Inspector Window

The **toolbar** provides access to the most essential working features. On the left it contains the basic tools for manipulating the scene view & the objects within it. In the centre are the play & pause controls. The buttons to the right give you access to your Unity Cloud Services & your Unity Account, followed by a **layer** visibility menu, and finally the editor layout menu which provides some alternate layouts for the editor windows, and allows you to save your own custom layouts.

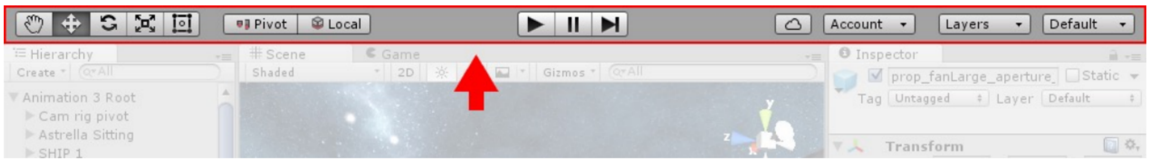


Figure 6: The Toolbar

1.4.1 The Game Loop

At their core, games operate a **game loop**, although game engines somewhat hide this from you. It operates at 60fps (or more), deal with inputs received asynchronously via events (e.g. keyboard or network data) or polled for right now, process game objects such as move the physics simulation (if any) forwards or move objects by physics simulation or direct control, redraw, wait (maybe) or run at maximum obtainable speed (maybe) but don't block the main thread.

1.4.2 The Unity API

Luckily, everything is not just drag-n-drop. We can write C# code to make the game work. Much of our code will involve the manipulation of the Unity API classes from the `UnityEngine` & `UnityEngine.UI` namespaces. Of course, we can also do anything else supported by the core C#/.NET library such as file handling, networking, collection classes, etc.

1.4.3 Component-Based Architecture

2nd generation game engines such as Unity3D use **Component-Based Architecture (CBA)**, which suits game logic very well. It's different to classic OOP as it's based on the **Composition** rather than **Inheritance** principle. Every

entity consists of one or more **components** which add additional behaviour or functionality. The behaviour of an entity can even be changed at runtime by adding or removing components. This eliminates the ambiguity problems of inheritance class-hierarchies that are difficult to understand, maintain, & extend. Each component is essentially a separate software object but all attached to some higher-level `GameObject` which typically equates to an actual onscreen character, enemy, bullet, effect, vehicle, etc.

Using composition rather than inheritance can be somewhat of a mantra, and not correct in all cases. Code re-use is often (incorrectly) considered to be the main principle of inheritance. In fact, designing hierarchical taxonomies of classes is the most appropriate reason to use inheritance. Composition is better when you would otherwise get tangles up in the murky world of multiple-inheritance. Composition is excellent for code re-use.

2 Lab Session 1

Import a model of Mars (and a `texture.jpg` for it). Create a simple demo which rotates Mars using the built-in physics engine. We could also take direct control of the Mars object and rotate it ourselves, a little bit on each frame, but here we're letting the physics engine do the work. No need to submit anything for this lab session as it's not graded.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameManagerScript : MonoBehaviour {
    public GameObject marsObject;
    // Use this for initialization
    void Start () {
        Camera.main.transform.position = new Vector3 (0f, 0f, -100f);
        Camera.main.transform.LookAt (marsObject.transform);
        // use the physics engine to rotate Mars
        // before this can run, you need to manually add a rigid body with 0 angular velocity and
        ↪  no gravity in the UI
        marsObject.GetComponent<Rigidbody>().AddTorque (new Vector3(0f,20f,0f));
    }
}
```

Listing 1: Lab 1 Code

```
public class GameManager : MonoBehaviour {
    // inspector settings
    public GameObject marsObject;
    void Start() {
        marsObject.transform.position = new Vector3(0,0,0);
        Camera.main.transform.position = new Vector3(0,0,100);
        Camera.main.transform.LookAt(marsObject.transform);
    }
    void Update() {
        // programmatically rotate Mars each frame:
        marsObject.transform.Rotate(new Vector3(0,10*Time.deltaTime,0));
    }
}
```

Listing 2: Alternative Lab 1 Code

3 Key Concepts & Classes

3.1 `MonoBehaviour`

`MonoBehaviour` is the base class from which Unity C# scripts normally derive. This hooks the class into the Game Loop so that it automatically receives calls to specific methods at specific times. It also provides various other useful Unity methods that are called by the game engine in specific situations, including but not limited to:

- `Start()`
- `OnDestroy()`
- `Awake()`
- `Update()`
- `FixedUpdate()`
- `LateUpdate()`
- `OnDisable()`
- `OnEnabled()`
- `OnBecameInvisible()`
- `OnBecameVisible()`
- `OnCollisionEnter()`
- `OnCollisionExit()`
- `OnCollisionStay()`
- `OnTriggerEnter()`
- `OnTriggerExit()`
- `OnTriggerStay()`
- `OnMouseEnter()`
- `OnMouseExit()`
- `OnMouseDown()`

In games, it is often useful to be able to execute code at programmer-controlled intervals (perhaps much less often than every frame), or at some specified time in the future. In Unity `MonoBehaviour`, the methods `Invoke()` & `StartCoroutine()` relate to this. We will elaborate on these later. `MonoBehaviour` also provides several important data members such as:

- `enabled` (Boolean).
- `gameObject` (GameObject).
- `transform` (Transform).
- `name` (String) (name of `gameObject`).

Finally, there are some more methods provided which are useful for manipulating `GameObjects` and their components:

- `SendMessage()`
- `BroadcastMessage()`
- `SendMessageUpwards()`
- `GetComponent()`
- `GetComponentInChildren()`
- `GetComponentInParent()`
- `GetComponents()`
- `GetComponentsInChildren()`
- `GetComponentsInParent()`
- `GetInstanceID()`

Recall that `GameObjects` can contain many independent scripts (components), each inheriting from `MonoBehaviour`. Each script/component is an instance of a class. A `GameObject` is therefore composed of multiple software objects. This is called a "component-based system" and is often seen as a superior approach to object-oriented class hierarchies – better for code re-use & isolation of functionality.

References:

- `https://docs.unity3d.com/ScriptReference/MonoBehaviour.html`
- `https://docs.unity3d.com/Manual/ExecutionOrder.html`

3.2 `GameObjects` in Unity

`GameObject` is the base class for all entities that exist in a Unity scene. As discussed above, each `GameObject` has a collection of components attached, and each component is an independent class object inheriting from `MonoBehaviour`. The `GameObject` is a special class that *all* entities in the game are derived from and which you won't have to attach as a component. Some useful data members of `GameObject` include:

- `activeInHierarchy` (Boolean).
- `transform` (Transform).

- `tag` (Tag type as defined in the editor).

Some useful methods of `GameObject` include:

- `AddComponet()`.
- `SendMessage()` etc. (same as the method in `MonoBehaviour` classes).
- `GetComponent()` etc. (same as the method in `MonoBehaviour` classes).
- `SetActive()`.

Some useful static methods of `GameObject` include:

- `Find()`.
- `Destroy()`.
- `Instantiate()`.

3.2.1 Getting References to `GameObjects` at Runtime

For a specific `GameObject`:

- Have it referenced as a public member of the script that needs it, and associated at design-time in the Inspector. (This is what we did in the first lab for the Mars object, a reference to which was needed by the `GameManager` class).
- Use `GameObject.Find("<name>")` to find it by name. This is somewhat inefficient, so don't do this every frame.
- Use `GameObject.FindGameObjectsWithTag("<tag>")` to find all game objects with a specified tag. This returns an array of `GameObjects`.

3.2.2 Prefabs

When you have created a `GameObject` in the hierarchy (at design time), added components to it, and set their various public values, you can drag the `GameObject` into the Assets Window to make a **prefab**, essentially a template of that exact object with its settings.

This approach means that you can make `MonoBehaviour` scripts for monsters, guns, etc. and then make a separate prefab with different settings for each actual type of monster or gun you need in your game. Prefabs are therefore useful for creating & editing your game's data.

3.2.3 Runtime Instantiation & Destruction of `GameObjects`

So far, we have created all of the `GameObjects` that we need in the hierarchy at design-time (i.e., before starting the game). Often, we need to instantiate & destroy some or all of our game objects at runtime, e.g. bullets, enemies, explosions, etc. Assuming that we have a **prefab** in our Assets, and that it is located in a directory called `Resources` we can do the following. Here, `prefabName` is the name of our asset prefab (as a String):

```
GameObject go = Instantiate(Resources.Load(prefabName));

// later on, assuming we  still have a reference to the object:
GameObject.Destroy(go);
// or
GameObject.DestroyImmediate(go);
```

Alternatively, you can instantiate an object by supplying an existing instantiated object as a template. E.g., in Lab 1 our `GameManagerScript` class had a `GameObject` member called `marsObject`, which is instantiated in the Inspector at design time, so you could use this to make another Mars:

```
GameObject otherMarsObject = Instantiate(marsObject);
```

Another alternative, perhaps the easiest of all, is to have a public reference to a `GameObject` and associate this with a prefab in the assets by dragging in the Inspector. Then you can use `Instantiate()` on that to instantiate a copy of the prefab into the scene at runtime.

3.2.4 Cameras

In computer graphics, **Cameras** are software objects that define a viewpoint in terms of position, orientation, field-of-view/zoom, etc. within the "virtual world". Mathematical projections are used to calculate how the 3D world should be displayed on the 2D camera surface. In Unity, Cameras are `GameObjects` which have a `Camera` component. Since they're `GameObjects`, you can add other components, manipulate their `Transform`, etc. just like any other `GameObject`. The `Transform` component provides methods that are very useful to Cameras (as well as other non-Camera objects) such as `LookAt()`.

`Camera.main` is a static reference to the "main" Camera whose view is being displayed to the screen. This Camera should have the tag `MainCamera`. Other Cameras can be used at the same time, e.g. to render to texture (e.g. mirrors, top-down minimaps) or to render to parts of the screen (e.g., split-screen multiplayer). You might also have multiple cameras in a scene and switch the active one during the game.

The `Camera` class provides methods for converting co-ordinates between Screen-space, Viewport-space, & World-space (more on these later).

- `CalculateFrustumCorners()`: Given viewport coordinates, calculates the view space vectors pointing to the four frustum corners at the specified camera depth.
- `ScreenPointToRay()`: Returns a ray going from camera through a screen point.
- `ScreenToViewportPoint()`: Transforms position from screen space into viewport space.
- `ScreenToWorldPoint()`: Transforms position from screen space into world space.
- `ViewportPointToRay()`: Returns a ray going from camera through a viewport point.
- `ViewportToScreenPointTransforms()`: position from viewport space into screen space.
- `ViewportToWorldPointTransforms()`: position from viewport space into world space.
- `WorldToScreenPointTransforms()`: position from world space into screen space.
- `WorldToViewportPointTransforms()`: position from world space into viewport space.

References:

- https://docs.unity3d.com/ScriptReference/Camera.html
- https://docs.unity3d.com/Manual/class-Camera.html

3.2.5 Transforms

A **Transform** defines the position, orientation, & scale of an object. Child objects have their own Transforms which are interpreted as a nested co-ordinate system, i.e. a modification of the parent's. This is a very powerful concept and is fundamental to the concept of a SceneGraph/hierarchy. In Unity, the `Transform` class has the following key members:

- `position` (Vector3)
- `localPosition` (Vector3)
- `rotation` (Quaternion)

- `localRotation` (Quaternion)
- `lossyScale` (Vector3)
- `localScale` (Vector3)
- `parent` (Transform)
- `right`: positive direction on the local x axis (Vector3, normalised).
- `up`: positive direction on the local y axis (Vector3, normalised).
- `forward`: positive direction on the local z axis (Vector3, normalised).
- `gameObject` (GameObject)

Methods of the `Transform` class include:

- `Rotate()`: Uses "Euler angles" (x, y, z) all in degrees.
- `Translate()`
- `TransformPoint()`: Transforms position from local space to world space.
- `InverseTransformPoint()`: Transforms position from world space to local space.
- `LookAt()`: (Vector3 point) turns so that the forward direction (i.e., positive z axis) faces the specified position.
- `RotateAround()`: (Vector3 point, Vector3 axis, float degrees).
- `SetParent()`

Reference: https://docs.unity3d.com/ScriptReference/Transform.html

3.2.6 Skyboxes

Skyboxes are rendered around the whole scene in the background in order to give the impression of complex scenery at the horizon. To implement a Skybox, create a Skybox Material in your Assets. Then, add it to the scene by using the Window → Rendering → Lighting menu item and specifying your Skybox material as the Skybox on the Scene tab.

Reference: https://docs.unity3d.com/Manual/class-Skybox.html

3.2.7 Keyboard Input

We will revisit more details about Keyboard/Mouse/Joystick/Touchscreen input later, but for now we will just say that `Input.GetKey()` returns true as long as a key is held down. E.g., this could be written in an `Update()` method:

```
if (Input.GetKey(KeyCode.LeftArrow)) {
    RotateCameraAroundMarsSomehow();
}
```

References:

- https://docs.unity3d.com/ScriptReference/Input.html
- https://docs.unity3d.com/ScriptReference/Input.GetKey.html

3.2.8 Static Member Variables

By declaring a member variable **`static`**, you're creating a single copy of the variable which is owned by the class rather than by any specific instance object. Each instance does not have its own unique copy, they share one. A common use case for static member variables is if you want to retain a collection of all instances of the class for quick recall.

3.2.9 Static Member Functions

By declaring a method `static`, you're creating a method that is called on the class itself, not an instance of it. This is very useful since you don not need to have a reference to an instantiated object from the class in order to call it. For obvious reasons, a static method only has access to static member variables of the class, while a non-static method (which is called on an actual instantiated object) has in addition, access to its own copies of non-static member variables.

3.2.10 The Singleton Pattern

A common design approach in Unity is to create single-instance **singleton** classes for various "management" roles, e.g. `GameManager`, `GUIManager`, `AudioManager`, `SaveGameManager`, etc. The typical approach is to create a `GameObject` in the hierarchy which holds the one-and-only instance of these components that will exist at run-time. One habit is to attach these to the Main Camera. Using statics can make the methods of these management singletons very easy & clean to access, e.g. consider the "instance" static member variable used here; it gives direct access to data attached to the instantiated singleton object, from anywhere in the game, via `GameManager.instance`

```
public class GameManager : MonoBehaviour {
    public static GameManager instance;

    void Start () {
        instance = this;
    }
}
```

3.3 Lab 2: Fear & Dread

In our first lab, we attached a `Rigidbody` to the Mars object, applied angular velocity to it using `AddTorque`, and let the game engine control the movement of it. The other way of moving game objectsis to directly manipulate their `Transform` using code (this was our approach in second year Java). In Lab 2, we'll be making Mars' two moons (Phobos & Deimos) orbit around it through direct manipulations of their Transforms. Your code for doing this would typically be written in the `Update()` method of some script (it could either be a script attached to the `GameObject` itself, or perhaps a script attached to a singleton "manager" object). It is important to consider the fact that `Update()` is probably not happening at fixed time intervals. You can multiply all movement code by `Time.deltaTime`.

4 Co-ordinate Systems

The most important co-ordinate system distinction in Unity is between the **Global** (or World) co-ordinate system and the **Local** co-ordinate system of each `GameObject`. Hence, to rotate the camera round the y-axis *as perceived by the camera* (i.e., the y-axis of its Local Co-ordinate system) we can use:

```
camera.transform.RotateAround(Vector3.zero, camera.transform.up, 50f * Time.deltaTime);
```

Or, to rotate it around the Global y-axis:

```
camera.transform.RotateAroudn(Vector3.zero, Vector3.up, 50f * Time.deltaTime);
```

The first `Vector3` argument defines the world point around which to rotate.

Some methods of the `Transform` class for translating from the local space of a transform to global space and vice-versa include:

```
// local space to global
public Vector3 TransformPoint(Vector3 position);
public Vector3 TransformDirection(Vector3 direction);
public Vector3 TransformVector(Vector3 vector);
```

```
// global to local
public Vector3 InverseTransformPoint(Vector3 position);
public Vector3 InverseTransformDirection(Vector3 direction);
public Vector3 InverseTransformVector(Vector3 vector);
```

4.1 Examples

To find the world-coordinate of a point which is 10 units "in front of" a spaceship (in terms of its own direction facing, and assuming this code is located inside a script attached to the spaceship):

```
Vector3 pt = transform.TransformPoint(new Vector3(0f, 0f, 10f));
// another way of doing the same would be:
Vector3 pt = transform.position + 10f * transform.forward;
```

To accelerate a spaceship forwards (assuming it has a `Rigidbody` and we're using physics, and we're doing this in a `FixedUpdate()` method on one of its scripts):

```
Rigidbody rigid = GetComponent<Rigidbody>();
rigid.AddForce(transform.forward * 200f * Time.fixedDeltaTime);
```

To get the direction & distance between two `GameObjects`:

```
GameObject go1, go2; // it's assumed that these are not nulls!

Vector3 difference, direction;
difference = go2.transform.position - go1.transform.position;
direction = difference.normalized;
float distance = difference.magnitude;
```

Iterating nested objects using `foreach (Transoform t in Transform)`:

```
public class Area : MonoBehaviour {
    // inspector settings
    public Transform innerWallsGroup;
    public Vector3 wallsPos;

    private List<GameObject> innerWalls = new List<GameObject>();
    private bool[] wallIsRaised = null;

    void Start() {
        foreach (Transform t in innerWallsGroup) {
            innerWalls.Add(t.gameObject);
        }

        wallIsRaised = new bool[innerWalls.Count];

        for (int i = 0; i < innerWalls.Coutn; i++) {
            innerWalls[i].transform.position = wallsPos;
            innerWalls[i].SetActive(false);
            wallIsRaised[i] = false;
        }
    }
}
```

4.2 Screen Space, Viewport Space, & World Space

Reference: https://docs.unity3d.com/ScriptReference/Camera.html

The `Camera` class provides methods for translating between three different "spaces", i.e. different ways of mapping the positions of things.

Each of these is stored as a `Vector3` (which is a `struct`). A **screen space** point is defined in pixels. The bottom-left of the screen is $(0, 0)$; the right-top is (`Camera.pixelWidth`, `Camera.pixelHeight`) . If you're using a `Vector3` rather than a `Vector2`, then the z position is in **world units** in front of the Camera, rather than in screen pixels. See also the `Screen` class: https://docs.unity3d.com/ScriptReference/Screen.html

A **viewport space** point is normalised and relative to the Camera. The bottom-left of the Camera is $(0, 0)$; the top-right is $(1, 1)$. Again, the z position (if any) is in world units in front of the Camera. E.g., this is appropriate for positioning GUI elements independent of actual pixel resolution of the screen.

A **world space** is defined in global co-ordinates (for example, `transform.position`. E.g., this is appropriate for `GameObjects` in the game world. `GameObjects` which are nested in the Hierarchy have both `transform.position` & `transform.localPosition`; `transform.rotation` & `transform.localRotation`, etc.

4.2.1 Typical Space-Translation Operations

Find where the mouse is in world co-ordinates to see if a `GameObject` is under the mouse:

```
// note that Input.mousePosition gives a Vector4 where the z component is 0
// the screen is 2D of course; the z component of the Vector3 supplied to ScreenToWorldPoint is a
↪ world-coordinate distance into the world.
Vector3 mousePosInWorld = Camera.main.ScreenToWorldPoint(Input.mousePosition);
```

Find a `GameObject` in screen-pixel coordinates to position a GUI item such as a healthbar above it:

```
GameObject targ; // assumed not to be null
Vector3 screenPos = Camera.main.WorldToScreenPoint(targ.transform.position);
```

```
// Here, we find the 3D position that's just above the enemy's head, and convert this to a
↪ vieweort position when viewed through the player's camera.
// Finally, this viewport position is converted to a pixel position as required by the GUI sprite.
public class MonsterManager : MonoBehaviour {
    void Update () {
        if (GameManager.gameState==GameStates.Playing) {
            if (numMonstersAlive==1) {
                // put the final enemy indicator (GUI object) above the final monster
                GameObject go = GUIManager.instance.finalEnemyIndicator;
                Monster m = allActiveMonsters[0];
                Bounds b = m.mycollider.bounds;
                Vector3 pos = new Vector3(b.center.x, b.max.y + 1f, b.center.z);
                Vector3 viewPos = Player.myPlayer.fpsCamera.WorldToViewportPoint(pos);
                if (viewPos.x<0f || viewPos.x>1f || viewPos.y<0f || viewPos.y>1f || viewPos.z<0f)
                    go.SetActive(false);
                else {
                    go.SetActive(true);
                    go.transform.position = new Vector2(viewPos.x*Screen.width,
                    ↪ viewPos.y*Screen.height);
                }
            }
        }
    }
}
```

Listing 3: The "Final Enemy Indicator" in DemonPit

5 Object Interactions with Colliders & Triggers

Objects which have `Colliders` & `Rigidbodys` will physically respond to impacts (i.e., they'll bounce off each other). Objects which have Triggers but not Colliders will **not** physically interact.

Related to the physics engine are two sets of `MonoBheaviour` methods which are called automatically when objects with Colliders or Triggers interact:

```
// these happen for Collider-to-Collider collisions
OnCollisionEnter(Collision coll);
OnCollisionExit(Collision coll);
OnCollisionStay(Collision coll);

// these happen for Trigger-to-Trigger collisions
OnTriggerEnter(Collider coll);
OnTriggerExit(Collider coll);
OnTriggerStay(Collider coll);
```

The `Collision` argument provides information on the game object that has collided with us, as well as additional information about the collision itself (e.g., the speed of impact). The `Collider` argument is simply a reference to the Trigger component that interact with ours. `OnCollision` refers to an actual physics interaction between two Colliders (e.g. `BoxCollider` or `SphereCollider` components). `OnTrigger` is the same thing except here the Collider has its Trigger flag set to `true`. Triggers enable game-logic interaction when you don't want physical collision responses.

Reference: `https://docs.unity3d.com/ScriptReference/Collider.OnCollisionEnter.html`

5.1 Lab 3: Asteroid Assault on Mars & its Moons

This lab builds on the work from the last two weeks; we already have Mars rotating, with Phobos & Deimos orbiting around it, and keyboard control of the camera. This week, we will be instantiating asteroids at runtime, and setting them moving (using physics). Asteroid which collide with anything will be destroyed, as will be asteroids which pass offscreen.

5.2 Triggers (Sensors)

Triggers are invisible components that are activated (triggered) when a character or other object passes inside them. They are a very common & useful mechanism in games. In Unity, Trigger is made by using a Collider with its Trigger property checked.

5.3 Coroutines

Coroutines are special functions that can pause their execution. They're very useful for delayed execution, making things happen in steps (with a defined delay between each step), or for waiting for something else to complete before executing. Coroutines are a very useful feature provided by the `MonoBehaviour` class. Their usefulness is because the code related to something occurring in the game is kept all in one place, even if that something takes a long time to occur, or occurs in stages. Coroutines are not threads, as they run in the same thread as the main Unity process, but they are much simpler to use than threads. In any case, most of the Unity SDK is not thread-safe. Coroutines are started with a call to `StartCoroutine()`, and they must have the `IEnumerator` return type. Technically, the way that these work is that Unity maintains a list of active Coroutines and on each game loop wakes up any that are now due to wake up.

When the coroutine is activated, it well execute until the next `yield` statement and then pause until it is resumed. Various values can be supplied with the `yield` statement, e.g. `yield return null` simply waits until the next frame.

```
private void Start() {
    Debug.Log("Start method");
    StartCoroutine(TestCoroutine());
    Debug.Log("Start method ends");
}

// this coroutine has a loop that runs as along as the calling object is active
private IEnumerator TestCoroutine() {
    Debug.Log("TestCoroutine");
    while(true)
    {
        Debug.Log("Here");
        yield return null;
        Debug.Log("There");
    }
}

private void Update() {
    Debug.Log("Update");
}

// Output should be:
/*
Start method
TestCoroutine
Here
Start method ends
Update
There
Here
Update
There
Here
*/
```

Some of the most useful Coroutine `yield` values include:

- `yield return null;` – the coroutine is continued the next time that it is eligible, normally on the next frame.
- `yield return new WaitForSeconds(3f);` – causes the coroutine to pause for a specified time period (three seconds in this example). Be aware of the garbage implications of this.
- `yield return StartCoroutine(OtherCoroutine());` – waits until the other coroutine has run to completion before the yielder is resumed.
- `yield break;` – stops the coroutine and exits (i.e., this is the equivalent of a `return` statement in a normal method).

Be aware of changes during the `yield` time. You always have to be careful with asynchronous programming: something important may have changed during the "down time" of a coroutine.

Reference: https://unitygem.wordpress.com/coroutines/

5.3.1 Invoke & InvokeRepeating

`Invoke` & `InvokeRepeating` provide a simpler way of running a complete method some time in the future, either as a one-off or a repeating call.

```
public class ExampleScript : MonoBehaviour
{
    // Launches a projectile after 2 seconds

    Rigidbody projectile;

    void Start() {
        Invoke("LaunchProjectile", 2f);
    }

    void LaunchProjectile()
    {
        Rigidbody instance = Instantiate(projectile);
        instance.velocity = Random.insideUnitSphere * 5f;
    }
}
```

To invoke repeatedly:

```
InvokeRepeating("LaunchProjectile", 2.0f, 0.3f);
```

5.4 Layers

Layers allow you to group game objects into categories. They are can be found at Edit → Project Settings → Tags and Layers. You can then decide which layers interact with each other in the physics simulation. You can also use an object's Layer to decide what happens when it collides with another object. You can also apply raycasts (see later) on specific objects only, e.g. to determine whether a monster can see a player, you would cast a ray between the position of the monster's head and the player's head, but only let the raycast consider walls/floors, i.e. it would ignore any creatures in the way.

Don't confuse Layers with Sorting Layers: the latter is for deciding the draw order of Sprites.

Reference: https://docs.unity3d.com/560/Documentation/Manual/Layers.html

6 Physics

6.1 Rigidbody

References:

- https://docs.unity3d.com/Manual/RigidbodiesOverview.html
- https://docs.unity3d.com/ScriptReference/Rigidbody.html

A `Rigidbody` is the main component that enables phyiscal behaviour for a `GameObject`; it puts the `GameObject` under the control of the physics engine. The object will respond to gravity if it has a `Rigidbody` component attached, (provided that the `useGravity` field is not set to `false`. If one or more Collider components are added, the `GameObject` is affected by incoming collisions (from other Colliders) according to its shape, mass, momentum, linear, & angular velocities, as well as Physics Materials which defined bounciness & friction.

Since a `Rigidbody` component takes over the movement of the `GameObject` that it's attached to, you normally shouldn't try to move it from a script by changing the Transform properties such as position & rotation; instead you should apply forces to push the `GameObject` and let the physics engine calculate the results. When a `Rigidbody` is moving slower than a defined minimum linear or rotational speed, the physics engine assumes that it has come to a halt. When this happens, the `GameObject` does not move again until it receives a collision or force, and so it is set to "**sleeping**" mode. This optimisation means that no processor time is spent updating the `Rigidbody` until the next time that is "awoken", i.e. set in motion again. Sleeping can also be useful as it removes small "jitters" that may happen due to inaccuracies in the physics simulation. In your scripts which apply physics forces to Rigidbodies, use the `FixedUpdate()` method rather than the `Update()` method, since `FixedUpdate()` is synced with the physics simulation updates.

`Rigidbody` properties:

- `public float drag`
- `public float angularDrag`
- `public float mass`
- `public Vector3 velocity`
- `public Vector3 angularVelocity`
- `public Vector3 centerOfMass` (offset from Transform centre).

`Rigidbody` methods

- `public void AddForce(Vector3 force)` (`force` is a vector in world co-ordinates applied through the centre of mass of the `Rigidbody`.
- `public void AddRelativeForce(Vector3 force)` (`force` is a vector in local co-ordinates).
- `public void AddForceAtPosition(Vector3 force, Vector3 position)`

6.2 Colliders

Reference: https://docs.unity3d.com/Manual/CollidersOverview.html

Collider components define the shape of an object for the purposes of physical collisions. Colliders are invisible at runtime, and do not need to be the exact same shape as the object's mesh; a rough approximation is of the shape is often more efficient, and is indistinguishable in gameplay. The simplest (& least processor-intensive) colliders are the **primitive** collider types, which all inherit from the superclass `Collider`. In 3D, these are the `BoxCollider`, `SphereCollider`, & `CapsuleCollider`. In 2D, these are `BoxCollider2D` & `CircleCollider2D`. Any number of these can be added to a single object to create composite colliders to reasonably approximate a 3D model.

If you need more accuracy (at increased processor cost), use **`MeshCollider`** which accurately matches the 3D graphical model (the polygonal mesh). A `MeshCollider` will be unable to collide with another `MeshCollider` unless you mark it as `Convex` in the inspector. This will generate the collider shape as a "convex hull" which is like the original mesh with concavities filled in. The general rule is to use mesh colliders for static scene geometry (walls, ground, etc.) and to approximate the shape of moving objects using composite primitive colliders.

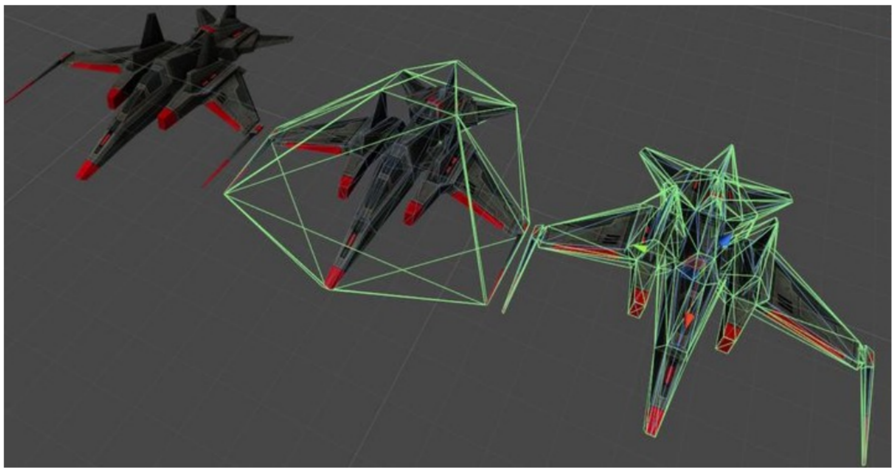


Figure 7: `MeshCollider`: Convex Hull vs Non-Convex

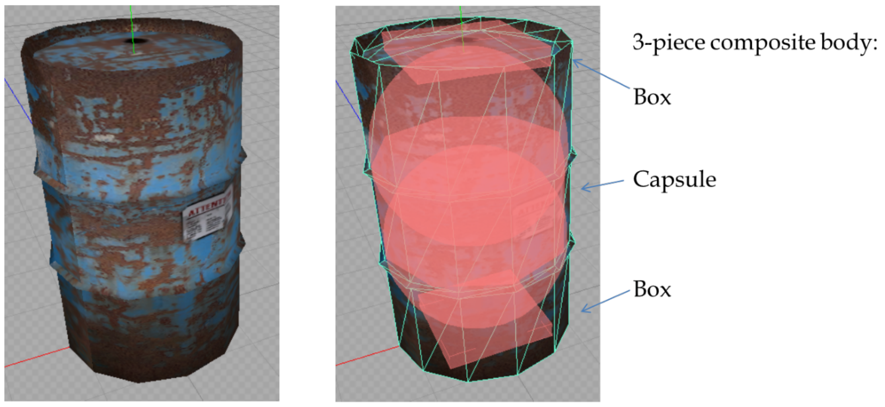


Figure 8: Composite Collider

6.2.1 Collisions

Reference: `https://docs.unity3d.com/ScriptReference/Collision.html`

Any object with a Collider receives messages from the physics engine when it collides with other Colliders. Any script on the object may choose to respond by implementing the following methods:

- **`void`** *`OnCollisionEnter`*`(Collision collision)` (the `Collision` object contains information about contact points, impact velocity, etc.)
- **`void`** *`OnCollisionExit`*`(Collision collision)`
- **`void`** *`OnCollisionStay`*`(Collision collision)`

The `OnCollision` methods receive objects of type `Collision`, which contain useful information such as:

- `collider`: The `Collider` object that was hit.
- `contacts`: The contact point(s) generated by the physics engine as an array of `ContactPoint` structs.
- `gameObject`: The `GameObject` whose collider was collided with.
- `relativeVelocity`: The relative linear velocity of the two colliding objects as a `Vector3`.
- `rigidbody`: The `Rigidbody` that was hit; this is `null` if the object had a collider but no `Rigidbody`.

6.2.2 `ContactPoint` Struct

Reference: https://docs.unity3d.com/ScriptReference/ContactPoint.html

`Collision` objects contain arrays of `ContactPoint` structs in their `.contacts` member variable. The member data of the `ContactPoint` struct is as follows:

- `point`: The point of contact in `Vector3` world co-ordinates.
- `normal`: The surface normal at the contact point in `Vector3`.
- `otherCollider`: The other `Collider` in contact at the point.
- `thisCollider`: The first collider in contact at the point (useful if a `GameObject` has more than one `Collider` and we need to know which one).

These fields allow us to operate on the object that we hit and to create accurate special effects etc. at the point of contact, rotated according to the surface normal at that point.

6.2.3 `PhysicMaterial`

When Colliders interact, their surfaces need to simulate the properties of the material that they are supposed to represent. For example, a sheet of ice will be slippery while a rubber ball will offer a lot of friction and be very bouncy. Although the shape of colliders is not deformed during Collisions (hence the term “rigid body physics”), their friction & bounce can be configured using `PhysicMaterials`.